移动计算 时序逻辑 形式方法 计算模式 (6)

计算机科学2000Vol. 27№. 6

22-27

基于时序逻辑的移动计算的形式方法分析*)

Analyzing the Temporal Logic Based Formalism for Mobile Computing

TP301.6

魏 峻

(中国科学院软件研究所计算机科学开放研究实验室 北京100080)

(中国科学院软件研究所对象技术中心 北京 100080)

Abstract At present, mobile computing is widely considered as a new computing paradigm supported by advanced computational technologies. One of main characteristics of this paradigm is the ability to dynamically change the binding for hardware and/or software components, that is mobility. Although there are many languages appeared to announce supporting mobile computing, the requirements and features of this paradigm are still scarcely recognized. So a lot of research is being expanded on formal models and methods for mobile computing. In this paper, we analyze a temporal logic based formalism for mobile computing—— Mobile Unity. At first, we summarize the features of mobile computing, and generalize that mobile systems should be decoupled, strongly-autonomous, context-dependent and owning new requirements for location-transparency. Then we introduce Mobile Unity, its extensions for UNITY from syntax to computational semantics, specially analyze the correspondence between the language structures of Mobile Unity and the abstraction of mobile features, such as location, mobility, transient interactions etc. At the end, we conclude its some deficiency for supporting mobile computing.

Keywords Mobile computing, Temporal logic, UNITY, Location, Mobility, Transient interaction

现今,移动计算被广泛认为是一种由计算新技术支持的新计算范型,其主要特征是计算组件能与计算资源动态改变绑定关系,表现出移动性。有两类关注移动特性的计算——mobile computing 研究基于无线网络的移动设备上进行的计算活动及相关问题;mobile computation 研究基于 WEB 的移动程序。多项研究表明,在设计层次可以一致地抽象这两类分布式计算,统一为移动计算[2,14]。

尽管有很多新技术和语言支持移动计算,如 Java、Telescrip、Obliq、Aglets、Pict、Facile、Oz、Jocaml 等,但目前对移动性带来的需求和特征还缺乏较完整和深入的认识,因而许多工作在抽象层次借助形式模型和方法来认识移动计算的特征。例如分布式和高阶 π 演算[11]以及相关框架——Ambient 演算[3]、Joint 演算[5]、Klaim[4]、Seal 演算[12]等,这些进程代数模型和演算提供了基于操作语义的规范描述和特性验证基础,它们能够很好刻画系统重配置(结构与行为)、移动 agent 和系统主机失败的影响,但没有解决好组件断连情况。由于面向实现层次,它们针对的是移动计算的"how"层面;而基于时序逻辑的方法强调系统级性质描述和验证,是面向抽象规范说明的"what"层次,例如移动 UNITY[10]基于 UNITY 并发模型[3],不仅扩充了 UNITY 逻辑,而且扩充了语言构造,可分别处理移动计算的"how"与"what"两个层面,既很好地刻画了系统重配置与组件断连(续连),又能进行移动计算系统性质验证。

1 移动计算的特征

移动计算范型支撑设施的特点,包括无线网络通信质量低、带宽波动、断连与响应时间长,WEB 地理位置自然分布以及分布式应用需求等,决定了移动计算系统结构是松耦合、强自治,计算环境依赖,以及有新的位置透明需求等特征。

松耦合强自治:无线通信的低带宽、频繁断连和高

*)本文部分得到国家重点基础研究发展规划项目(973)的资助。魏 峻 博士,主要从事软件形式化规范,移动 agent 计算形式理论研究。

响应时间决定了移动计算的系统体系结构是松耦合强自治风格。松耦合指组件间,组件与服务器断连或弱连接时仍继续工作,而且只要连接可用,就进行"机会性"连接。强自治是指计算组件必须有很好的自包含性和很强的主动性。移动计算强调计算实体的自治与封装,相互通信较少,极端情况是机器崩溃、失败或有意断连时仍要独自运作。

计算环境依赖:移动计算的特点决定了计算组件经常改变计算环境—位置,这通常需要相关服务支持。计算组件间连接与交互的动态性带来了类似开放式软件系统的许多不可预料问题。组件必须在无数可能出现的配置中正确地工作,而且必须在组件重配置时继续工作。

新的位置透明需求:分布式计算的理想目标是透明分布性,并利用已有的非分布性成果。但移动计算的以上特征决定了要实现该目标是更大的挑战。在高层实现透明分布,但在低层必须面对且解决移动计算特征带来的问题,例如,移动 IP 试图在 INTERNET 环境提供位置透明的通信能力,但其中的位置注册算法是为透明移动服务的,协议设计者必须面对移动,显式讨论位置与位置改变来给出正确的实现。

2 UNITY 概览

UNITY[1]是基于线性时序逻辑的形式框架程序,包括并行程序设计语言和基本的程序逻辑系统。UNITY 程序的执行模型是非确定、公平无限,交叠地选择赋值语句执行。每个执行步产生程序状态的一个原子转换。UNITY 程序包括三部分:Declare、Initially、Assign 部分。UNITY 逻辑是基于交替并发语义模型的线性时序逻辑,其证明系统是基于 Hoare 逻辑对并发程序性质(安全性和活动性)进行证明的逻辑系统。例如安全性质 p co q(指程序若在某状态满足 p,则任一赋值语句执行后导致的下一状态必须满足 q),表示为 p co q(∀ s:: {p}s{q}) ∧ p⇒q。活动性质 transient p 说明谓词 p 的成立最终会被破坏。在弱公平假设下,该性质定义为:transient p(∃ s:: {p}s{e¬p})。该性质可用于定义其他活动性质,如 p ensure q (p∧¬q co p∨q)∧ transient(p∧¬q)。ensure 算子表示若程序在某状态满足 p,它保持在该状态除非 q 被建立。ensure 表示简单的前进性质,证明更复杂的前进性质需归纳证明程序是通过一序列步骤到达目标的,这可用 leads-to() 算子表示,其细节见文[3]。

标准 UNITY 中最基本的构造机制是联合(union),多个程序的联合是所有程序变量的联合,所有赋值语句的联合—以公平原子交叠方式执行,初始条件是所有初始条件的交集。另一种方式是重叠—以同步语句而不是共享变量方式组合组件。对一个基础程序的重叠是这样进行的:加入新语句和变量,新语句不对原程序的原始变量赋值,每个新语句与原程序的某些语句同步。重叠机制维护历史变量,不改变原程序的行为,同时层次性构造系统。然而重叠很有限,通信只能单向进行。与联合一样,重叠本质上是静态组合,固定了组件间的关系。

3 移动 UNITY

移动 UNITY 是在充分分析移动计算特征和 UNITY 计算模型,证明逻辑基础上,扩充语言构造和证明逻辑建立的移动计算形式框架,其目的是提供一种方法能描述和推理松耦合且计算环境依赖的分布式程序的形式系统。其中,移动性通过给每个程序附着一个位置变量,控制位置变量值变化来表现。松耦合强自治通过分离进程名字空间,封装组件描述,且分离组件与组件间交互描述来体现。组件的动态交互(断连与续连)由暂态交互表示,其中关键概念是反应语句及其语义。

3.1 语言构造的扩充

一个移动系统的程序由 Description 部分+Component 部分+Interaction 部分组成。其中清楚地区分了参数化程序类型与系统组件,划分了程序名字空间,加入新变量 λ 表示程序当前位置,Description 部分是组件类型描述,它们在 Component 部分实例化,组件实例间的暂态交互在 Interaction 部分给出。

移动 UNITY 给出一些新程序构造,主要用于描述与计算环境相关的组件交互与协调,它们与原 UNITY 的程序构造机制是正交的,一般在 Interaction 部分给出。这些程序构造的语义体现在证明逻辑依赖的基础计算模型中,在程序描述层抽象掉了。

1)**标记** 给语句增加引用机制。标记的形式为 $n::s_i$。程序中标记都是唯一的、其主要作用是用于构造禁止语句。

2)**额外语句** *when* 作为加强的语句卫士项。如 receiver. bit := sender. bit *when* sende λ=receiver. λ,表示当 receiver 与 sender 在同一位置时,sender. bit 的值拷贝到 receiver. bit。*when* 谓词可以考虑表示任意因素。例如,组件间的距离或其他组件的出现。

3)**事务语句** 语句串行执行的一种形式。由一系列赋值语句以指定顺序调度组成,其间没有其他非反应语句交替。单个赋值语句可看作单元素事务,它与事务都是以弱公平方式选择,或以原子动作执行或作为连续原子动作序列的部分执行,事务的形式为$\langle s_1, s_2, \cdots, s_n\rangle$,$s_i$ 是赋值语句。当调度器选择该事务语句执行时,必须是先 s_1,再 $s_2\cdots$。在没有反应语句的情况下,该

语句的效果等同程序状态的一个原子转换。

4)禁止语句 当不期望在全局计算环境中出现某语句执行时，禁止语句提供了一种既加强已有语句卫士项又不修改原语句的机制，使用该构造可以约束 UNITY 非确定性调度器。禁止语句的形式为 *inhibit* n *when* p，n 是语句的标记，p 是谓词，

5)反应语句 用专断的终止性计算扩充赋值语句的机制。它以一致的方式扩充所有赋值语句。所有反应语句优先级高于一般语句。一个程序的所有反应语句形成一个终止性子程序，它在每个赋值语句(包括事务的)之后被调度执行直到没有语句再产生影响，形式上这称为集合的不动点。谓词 FP(R)可以计算出来。反应语句形式为 s *reacts-to* p，可理解为通过反应从句加强卫士项的赋值语句。例子 receiver. bit:=sender. bit *reacts-to* sender. λ=receiver. λ 表示当 sender 与 receiver 在同一位置时，sender. bit 的值任意改变都立即反应到 receiver. bit，

```
System Sender-Receiver-Timer
 Program sender(i) at λ
 Declare bit:boolean
 []word:array[0  N-1] of boolean
 []c,t,sndstamp:integer
 initially λ=SenderLoc(i)
 assign
  trans:: bit,c:=word[c],c+1 if c<N and t>=sndstamp
          +dlta*c
  []new::=word,c,sndstamp:=New Word(),0,t if c>=
         N
  []timer::=t:=t+1 if t<sndstamp+dlta*c+dlta/4
end
Program receiver(j) at λ
Declare bit:boolean
 []buf:array[0…N-1] of boolean
 []c,t,recvstamp:integer
 assign
  recv::buf[c],c:=in,c+1
       if c<N and t>=recvstamp+dlta*c-dlta/2
  []zero::c,recvstamp:=0,t reacts-to bit=1 and c>=N
  []timer::=t:=t+1 if t<recvstamp+dlta*(c-1)-dl-
           ta/4
  []move::λ:=buffer reacts-to ValidLoc(buf) and c>=N
end
Component
  sender(1)[]sender(2)[]receiver(0)
Interaction
 Receiver(j). bit:=sender(i). bit reacts-to sender(i) λ=re-
              ceiver(j). λ
 inhibit sender(i). timer
   when sender(i). t-sndstamp>receiver(j). t-recvstamp
       and sender(i). λ=receiver(j). λ
 inhibit receiver(j). timer
   when receiver(j). t-recvstamp>sender(i). t-sndstamp
       and sender(i). λ=receiver(j). λ
 end
```

这种两阶段的计算模式，即连续的赋值语句被反应语句截断，初看似乎不合理，因为用包含许多复杂动作的反应语句可能写出完全不现实的系统规范，但用标准 UNITY 也会写出不现实的 UNITY 程序，移动 UNITY 追求灵活而强大的表达能力，而不是表示预定约束。有效实现是设计者的责任，关键在于控制运用已有的程序构造表示法。

3.2 证明逻辑的扩展

移动 UNITY 也扩充了 UNITY 逻辑系统来证明移动系统程序的性质。移动 UNIYT 扩充后的程序语句组成计算模型中的原子状态转换。其中反应式扩充的语句表示为 s^*，非反应语句集合为 $\mathcal{SS}$。Hoare 逻辑程序证明基本形式对反应式扩充仍适用 $\{p\}s^*\{q\}$，因此，co，ensure 性质表示为 $p\ co\ q \equiv (\forall s\in \mathcal{SS}::\{p\}s^*\{q\})$，$p\ ensure\ q \equiv (p\wedge\neg q\ co\ p\vee q)\wedge(\exists s\in\mathcal{SS}::\{p\wedge\neg q\}s^*\{q\})$。$\{p\}s^*\{q\}$的推导规则以假设-结论形式给出，对单元素事务的非反应语句 s 为：

$$\frac{p\wedge t(s)\Rightarrow q,\ \{p\wedge\neg t(s)\}s\{H\},H.H\ FP(\mathcal{SF}),\ stable\ I\ in\ \mathcal{SF},H\Rightarrow I,I\wedge FP(\mathcal{SF})\Rightarrow q}{\{p\}s^*\{q\}}$$

其中 H 表示语句 s 在其未被禁止状态执行后的状态成立的谓词，I 为经过所有反应语句 $\mathcal{SF}$ 执行后成立的不变式，t(s)为 s 语句的 inhibit 从句所有 when 谓词的析取。

对形式为$\langle s_1;s_2;\cdots;s_n\rangle$的语句，可以递归使用下面推导规则：

$$\frac{\{a\}\langle s_1;s_2;\cdots;s_{n-1}\rangle^*\{c\},\{c\}s_n^*\{b\}}{\{a\}\langle s_1;s_2;\cdots;s_n\rangle^*\{b\}}$$

其中的 c 可猜定或从 b 导出，

其他 UNITY 推导工具保持不变，这样就可以从这些简单的原语导出复杂的程序性质。

3.3 移动计算特征的语言抽象

位置 移动 UNITY 表示的移动计算系统是在组件暂态连接与交互的情况下运作的。连接与交互由组件当前位置决定，因此，位置是模型的一部分。模型中没有对位置作类型限制，可以是离散或连续的，即可对应移动平台的物理坐标，也可是移动 agent 所在网络或内存地址。但在语言中是由位置变量 λ 刻画的，它的类型可以按需求变化。程序中必须显式地控制位置，一般是根据程序状态设置 λ 的新值，

移动 程序通过设置位置变量非显式地刻画移动，一般形式是类似语句 λ:=NewLoc(λ)，表示一个执行程序迁移到新位置，例如代码(agent)移动即可这样表示。即使程序没有对自己位置加以控制，但内部调度器仍能通过公平选择执行内部赋值语句来进行移动组件，任何关于移动的限制应在位置控制语句中反映，

配置 完整的移动 UNITY 程序构成一个系统配置描述。配置变化是隐式的，即组件交互约束条件的变化引起暂态连接的变化，带来交互连接的配置重构。

交互 移动 UNITY 基于共享内存范型，给出了两类暂态交互机制——暂态共享与暂态同步来表示移动组件频繁断连和系统松散组合的特征，它们也是移动组件的组合机制。在移动 UNITY 的新语言构造中，反应语句捕捉了中断处理语义，它能表示局部和非局

部动作的同步执行；禁止从句捕捉了依赖关系处理语义。这两类语句都表示了跨单个组件界限的调度约束。额外 When 从句能加进语句组合，描述组件间条件异步数据传送语义。这些语句构成了表示组件交互的基本协调语言，使用它们可构造丰富的交互抽象，包括 UNITY 式共享变量、位置依赖的交互和基于时钟的同步。

(1)暂态共享　移动环境中的两个移动程序的变量是不相连的，这由名字空间反映。但若在程序的 Interaction 部分加进反应语句，则独立程序的变量就能呈现出共享变量的特征，即对同一位置的两组件中的一个变量的改变可立即反应到另一个，反应语句的语义保证传播是原子的。移动 UNITY 中使用暂态共享变量机制表示这类交互，它是用一种协调结构表示的。

≈关系。例如一个 laptop 与 printer 通过无线介质连接。当 laptop 在 printer 范围内，则它改变 printer 的打印队列。这表示为各自队列的暂态共享：laptop.queue≈printer.queue *when* laptop.λ=printer.λ_c≈关系可用传播值变化的反应语句形式地定义。由于共享是双向的，因此，使用两个反应语句表示双向值传播，而且使用了检测变化和选择性地传播新值的机制。单向暂态共享表示为：A.x→B.y *when* p。说明 A.x 被 B.y 暂态读取。使用反应语句给出的语义：

$B.y, B.y_{A.x}, A.x_{B.y} := A.x, A.x, A.x$ *reacts-to* $A.x \neq A.x_B \wedge p$

$A.x_{B.y} := A.x$ *reacts-to* $\neg p$

其中，$A.x_B$ 表示 A.x 的前一历史状态，$A.x \neq A.x_B$ 代表检测变化机制。

双向的暂态共享表示为：A.x≈B.y *when* p。其语义为 A.x→B.y *when* p，B.y→A.x *when* p。

该表示在保持连接时不会出现不一致，但在组件断连又续连的情况，就会出现状态不一致。因而在≈关系中引入由设计人员指定值改变策略的机制，其中加入 *engage* 从句表示：在断连又续连时共享变量的值；加入 *disengage* 从句表示：断连时共享变量的值。

A.x≈B.y *when* p *engage* e *disengage* d1,d2

disengage 一般在实际中不重要，因为断连是不可预料的。但对于描述有目的断连是很有用的机制，由于可预料断连几率很小，提供 *disengage* 概率也很小。

虽然暂态共享结构是两变量间的关系，但由于所有 *reacts-to* 语句执行到不动点，因而组合多变量，实现多组件暂态交互是很自然的。

暂态共享抽象是管理并发移动系统复杂性的有效结构，它提供了表达高度松耦合且环境依赖的系统组合的机制，该抽象很适于描述低层次的无线通信。

(2)暂态同步　上面给出了移动组件间共享状态的新抽象，它是暂态且位置依赖的，其中参与组件异步执行。但是，在许多分布系统模型中，语句同步执行是核心。例如，CSP 模型[6]的计算由静态串行进程进行，通信经过阻塞、非对称、同步、双方交互完成(输入/出命令)，I/O 自动机模型[7]经过同名的输入/出动作的同步完成通信。UNITY 中语句同步也是模型的一部分，同步执行用重叠表示，但是重叠在两个重要方面受到限制。第一，重叠系统是静态定义的。同步关系在系统的执行过程固定不变。而移动环境需要有动态指定变化和位置相关式同步的能力，从而使同步参与方能随计算的演进加入/退出同步关系。第二，重叠是非对称关系，一个程序包含另一程序，不允许被重叠方与原始程序通信。这种限制不适于移动计算领域，因为这种环境中的程序可能希望用同步进行双向通信，双方都获得服务。

移动 UNITY 采用类似 UNITY 的重叠机制刻画移动计算领域的同步，但使用一种协调结构，其中包含两个独立程序的语句。关键思想是当条件成立时，将两个语句合并为一个原子动作，$\rceil\lceil$ 表示语句同步算子。其描述的交互也是暂态、位置依赖的。

暂态语句同步由低级原语表示，基本思想是每个语句应对其他语句的选择反应继而执行，这样两个语句才可在同一原子步里执行，因此语句基本形式分开了语句选择与实际执行部分。例如，语句 A.s 的形式如下：

A.s driver :: (A.s_{phase} := GO; A.s_{phase} := IDLE)

A.s.action | A.s_l := false *reacts-to* A.s_{phase} = GO ∧ A.s_r

A.s_l := true *reacts-to* A.s_{phase} = IDLE　　(a)

非反应语句 A.s.driver 与其他非反应语句一起公平选择执行。在完成 A.s_{phase} := GO 后，A.s.action 会反应执行。这样描述一个语句就可以访问、控制语句选择与执行过程的关键部分。两个语句的同步简单地表示为阶段辅助变量的共享，表示为：

A.s $\rceil\lceil$ B.t *when* r　A.s_{phase} ≈ B.t_{phase} *when* r　　(b)

由于共享的传递性与多方共享，可以表示多方同步。

协选择表示两个语句同时被选择来执行。但(a)(b)表示的语义没有真正保证语句的同时执行，只是 R 中的某种交替顺序，在没有语句改变已由其他语句赋值的变量的情况下，该执行方式等价于同时执行，但有时需要在旧状态下计算赋值语句的右部，而不是用其他语句设置过的值。对这样的情况，需将语句实际动作分为右部计算阶段和左部赋值阶段。

A.s driver :: (A.s_{phase} := LOAD; A.s_{phase} := STORE; A.s_{phase} := IDLE)

A.s.load | A.s_M := false *reacts-to* A.s_{phase} = LOAD ∧

$A.s_{if}$

$A.s.store \| A.s_{af} := false$ *reacts-to* $A.S_{phase} = STORE \wedge A.S_{af}$ (c)

$A.s_{it}, A.s_{if} := true, true$ *reacts-to* $A.s_{phase} = IDLE$

这样表示的语句仍通过阶段辅助变量同步。所有语句在LOAD阶段计算右部，在STORE阶段赋值左部，在IDLE阶段重设标志。这样阻止了任两个语句间的干扰。

还有一些情况，希望在同步执行阶段通信。这种策略是CSP、I/O自动机模型的核心。CSP用通道把值从发送方传到接收方。I/O自动机能从一个输出语句传送任意参数到所有同名的输入语句。下面构造表示I/O自动机式的同步。

$A.s.driver :: \langle A.s_{params} := EXP; A.s_{phase} := GO; A.s_{phase} := IDLE \rangle$

$A.s.action \| A.s_f := false$ *reacts-to* $A.s_{phase} = GO \wedge A.s_f$ (a)

$A.s_f := true$ *reacts-to* $A.s_{phase} = IDLE$

加入 $A.s_{params} := EXP$ 作为事务的第一个语句，就把输出参数绑定到辅助变量 $A.s_{params}$。类I/O自动机式的同步表示为：

$A.s \| B.t$ *when* r $A.s_{params} \approx B.t_{params}$ *when* r

$A.s_{phase} \approx B.t_{phase}$ *when* r

通过定制禁止从句，很容易使用语句卫士项指定形式多样的同步。

(1)协选择

coselect$(A.s, B.t, r)$ $A.s_{phase} \approx B.t_{phase}$ *when* r

(2)协执行 表示共存的组件的语句只有在卫士项都成立时才可执行。

coexecute$(A.s, B.t, r)$ $A.s_{phase} \approx B.t_{phase}$ *when* r

inhibit $A.s.driver$ *when* $r \wedge \neg(A.s.guard \wedge B.t.guard)$

inhibit $B.t.driver$ *when* $r \wedge \neg(A.s.guard \wedge B.t.guard)$

(3)排斥性协执行 要求断开连接时，语句不能独立执行。

xcoexecute$(A.s, B.t, r)$ $A.s_{phase} \approx B.t_{phase}$ *when* r

inhibit $A.s.driver$ *when* $\neg(r \wedge A.s.guard \wedge B.t.guard)$

inhibit $B.t.driver$ *when* $\neg(r \wedge A.s.guard \wedge B.t.guard)$

(4)排斥性协选择

xcoselect$(A.s, B.t, r)$ $A.s_{phase} \approx B.t_{phase}$ *when* r

inhibit $A.s.driver$ *when* $\neg r$

inhibit $B.t.driver$ *when* $\neg r$

在I/O自动机与CSP等模型中的动作协调是非对称的，动作分为输入和输出，参数从输出动作传送到输入动作。CSP强调两两幽会式同步，而I/O自动机能表示一对多同步，移动UNITY中同步是对称的，但适当地使用事务、变量共享与禁止，能表示各种形式的同步。例如，限定单向变量共享关系集合，将其中输出语句作为事务执行，输入语句简单地反应，可以模拟一对多带参数传递同步。通过设置标记来保证阶段变量至多只传播到一个输入动作，而第一个非确定地被选中的反应语句设置该标记，阻止传播继续发生，这样可以模拟幽会式同步。

松耦合强自治 移动UNITY类似UNITY采用"[]"组装组件，但语法上区分参数化组件类型和系统程序，通过组件类型封装组件描述，在COMPONENT部分进行组件实例化，INTER ACTION部分通过各种交互机制定义组件的松散耦合。移动UNITY结构化程序描述很容易平坦化为无结构描述，可以使用扩展的逻辑系统[10]来推理性质。

总结 移动UNITY是在充分分析移动计算特征和UNITY计算模型、证明逻辑基础上，扩充语言构造和证明逻辑建立的移动计算形式框架。移动UNITY通过划分名字空间、描述组件类型来结构化移动组件系统，分离组件行为与组件协调行为；引入位置变量，将组件移动简化为变量赋值语句，从而可以推理计算的方式推理移动；通过暂态交互(暂态变量共享和暂态同步)将组件的通信转化为组件的协调，刻画低级的通信行为，如断连与续连操作、非常容易。同时，通过暂态交互建立的约束性连接表示系统配置动态重构。这些已应用于物理和逻辑移动环境的组件暂态交互的高级描述[10]，移动IP形式规范描述和形式验证[9]、移动代码范型(COD,REV,MA)的规范描述和验证[13]，以及细粒度代码移动模型规范与验证[1]等，表现出较强的表示与推理能力。

但移动UNITY在语言级没有组件创建与消亡等操作，不能直接表示组件加入/移出引起的系统配置动态重构。其计算模型是面向程序级的执行语义，逻辑系统没有直接面向抽象系统层给出反映移动计算的如移动性、位置相关等特征。这些都与开发移动UNITY的目标密切相关，因为它主要是为刻画与移动设备相关的通信问题而设计的。移动UNITY的经验告诉我们，任何形式方法不可能在各个方面有效，开发一种形式方法必须针对具体的目标，着重解决好某个层面的问题。

参考文献

1 Cardelli L, Gordon A D. Mobile Ambients. In M. Nivat, editor, Foundations of Software Science and Computation-

al Structures, LNCS No 1378, Springer-Verlag, 1998. 140～155
2 Cardelli L. Abstractions for mobile computations. [Microsoft Research MSR-TR-98-34]. 1997
3 Chandy K M, Misra J. Parallel Program Design: A Foundation. Addison-Wesley, New York, NY, 1988
4 De Nicola R, Ferrari G, Pugliese R. KLAIM: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. on Software Engineering, 1998, 24(5)
5 Fournet C, et al. A Calculus of Mobile Agents. In: Proc 7th Int. Conf. on Concurrency Theory (CONCUR), LNCS 1119. Springer, 1996
6 Hoare C A R. Communicating sequential processes. Prentice Hall, 1985
7 Lynch N A. Tuttle M R. An Introduction to Input/Output Automata. CWI Quaterly, 1989, 2(3)
8 Mascolo C, et al. A Fine-Grained Model for Code Mobility: [Washington University Tech. Report WUCS-99-07]. 1999
9 McCann P J, Roman G-C. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. In: 2nd Intel. Conf. on Coordination Languages and Models. Springer-Verlag, 1997
10 McCann P J, Roman G-C. Compositional programming abstractions for mobile computing. IEEE Transcations on Software Engineering, 1998, 24(2): 97～110
11 Sangiorgi D. From π-calculus to Higher-order π-calculus and back. In: Proc. of TAPSOFT' 93, LNCS 668, Springer-Verlag, 1992
12 Vitek J, Castagna G. A calculus of secure mobile computations. In. Proc. of the IEEE Workshop on Internet Programming Languages, (WIPL), Chicago, 1998
13 Picco G P, et al. Expressing Code Mobility in Mobile UNITY. In: Proc. 6th European Software Eng Conf. (ESEC/FSE' 97), LNCS1301. Springer, 1997
14 魏峻，冯玉琳. 移动计算形式理论分析与研究. 计算机研究与发展

（上接第12页）

件的基本更改，使用安全设施的应用也要求更改来支持安全接口协议。如果可能的话，安全更新是较难的。安全能力的考虑应该在结构处理中做早期的考虑以避免基本的威胁及减少更新成本。有远见的设计者应该计划系统的进化，并且把安全设施进一步发展为商业技术和标准。

5 CORBA 与 Java 技术的结合[7]

CORBA 与 Java 技术的结合有更大的优势。

（1）写一次，到处运行。几个与 CORBA 相关的 JavaORBS 已经发表。这些 JavaORBS 可以写100%的纯 Java 客户/服务器程序，使它独立于平台、到处运行。

（2）强的安全模型。Java 的 applet "沙箱"（sandbox）安全模型和字节码验证总体上避免了有预谋的密码攻击。Java 安全 API 提供了 DSA（数字签名算法）的具体实现方法。

6 CORBA 产品

（1）IONA 公司的 Orbix[8]。IONA 公司是目前的 CORBA 系统软件和服务提供商，其标志产品 Orbix 是一个基于库的 CORBA 实现。IONA 的系列产品包括 OrbixNames（名录服务）、OrbixTrader（交易服务）、OrbixSecurity（安全服务）、OrbixSSL（实现 SSL[9]的 IIOP 加密传输）。其中 OrbixSecurity 提供了安全服务，OrbixSSL 实现 SSL 的 IIOP 加密传输。

SSL 是由 Netscape 公司提出的安全交易协议，是一种利用公开密钥技术的工业标准，被用于 Netscape Communicator 和 Microsoft IE 浏览器，以及 IBM、Open Market 等公司提供的支持 SSL 的客户机和服务器，用以完成需要的安全交易操作。SSL 提供加密、认证服务和报文完整性，其安全性服务对用户来说尽可能透明。

（2）Inprise 公司的 VisiBroker。其中 VisiBrokerSSLPack3.2实现了 IIOP 加密传输。

参 考 文 献

1 汪芸，顾冠群. CORBA 技术综述. 计算机科学，1999，26(6)：1～6
2 CORBA successful stories, Object Management Group. Available at: http://www.corba.org/
3 CORBA Object Management Group responsible for the Open CORBA Standard Specification. Available at: http://www.omg.org
4 贾晶，陈元，王丽娜. 信息系统的安全与保密. 清华大学出版社，1999. 99～103
5 Dowd P W, Machinery J T. Networking Security: It's Time To Take It Seriously, Computer, Innovative Technology for Computer Professionals, 1998, 31(9): 24～48
6 Vinoski S. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communication Magazine, 1997, 35(2): 25～55
7 Curtis D, et al. White Paper of OMG, 1996
8 O'Toole A. Making software work together. Orbix Journal, 1998
9 Rubin A D, et al. A Suvey of Web Security, Computer, Innovative Technology for Computer Professionals, 1998, 31(9): 34～41